Lab08: Python System Scripting and Web Parsing

This lab gives practical examples to investigate System Programming, some basic Web page scraping and parsing, and calling external programs.

1. Web Page Analysis using Regular Expressions

This exercise is an essential component of the coursework.

In an earlier lab, you wrote a script webpage_get.py that reads the content from a webpage using the wget() function.

If we were carrying out some reconnaissance on an organisation's website we may want to parse out any links to other webpages, and other useful data. To parse the web page hyperlinks, we can use a regular expression.

From the IDLE Shell, using **File>New Window**, create a script **webpage_getlinks.py**, based on the code below. Note this script uses our wget() function from your webpage_get.py script to fetch the webpage!

```
# Script:   webpage_getlinks.py
# Desc:     Basic web site info gathering and analysis script.
#           From a URL gets page content, and parses out hyperlinks.
# Modified: Oct 2017
# Nov18 (PEP8)
import sys
import re
import webpage_get


def print_links(page):
    ''' find all hyperlinks on the webpage input and print '''
    print('[*] print_links()')
    # regex to match on hyperlinks
    links = re.findall(r'\<a.*href\=.*http\:.+', page.decode())
    # sort and print the links
    links.sort()
    print(f'[+] {len(links)} HyperLinks Found:')
    for link in links:
        print(link)


def main():
    # temp testing url argument
    sys.argv.append(r'http://www.napier.ac.uk')

    # Check args
    if len(sys.argv) != 2:
        print('[-] Usage: webpage_getlinks URL')
        return

    # Get the web page
    page = webpage_get.wget(sys.argv[1])
    # Get the links
    print_links(page)
```

```
if __name__ == '__main__':
    main()
```

Test from the IDLE shell, by running the module using **Run>Run Module.**

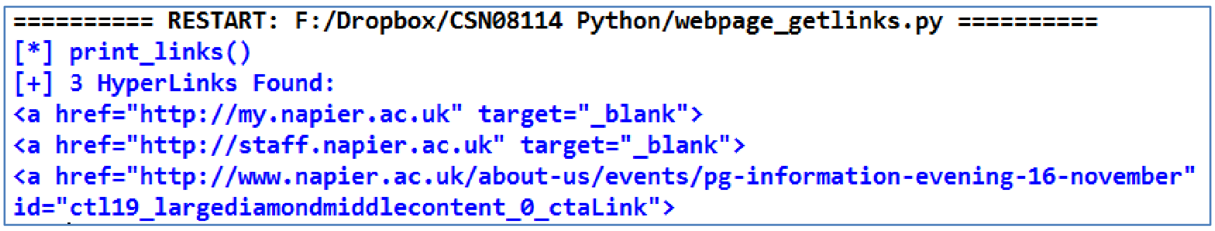
Hint: if you have trouble importing webpage_get, check that it is in the appropriate location!

Q: Have hyperlinks been matched against the pattern?

Q: How many links were found?   (top of output prints out number)

Q: Which statement allows us to use our wget() function?

You should get output similar to the following:



```
========== RESTART: F:/Dropbox/CSN08114 Python/webpage_getlinks.py ==========
[*] print_links()
[+] 3 HyperLinks Found:
<a href="http://my.napier.ac.uk" target="_blank">
<a href="http://staff.napier.ac.uk" target="_blank">
<a href="http://www.napier.ac.uk/about-us/events/pg-information-evening-16-november"
id="ctl19_largediamondmiddlecontent_0_ctaLink">
```

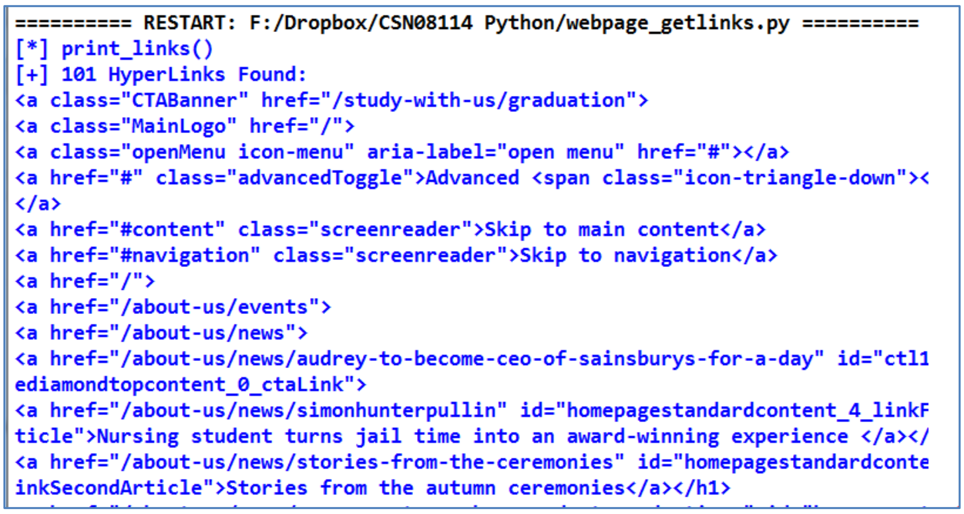
Q: Did the above find all the hyperlinks on the webpage?

Hint: Open the webpage http://www.napier.ac.uk/Pages/home.aspx in a browser and view the page source. In html, hyperlinks are created using the href= attribute inside an <a> tag. So we can search for them by searching for href= (use ctrl-f on the source).

Q: Inspect some of the hyperlinks that were not found. Explain why our regex did not find them.

So now that you have found out that our regex only matches absolute hyperlinks (those with full URL pathnames given), change the regular expression pattern until you get **all** hyperlinks.

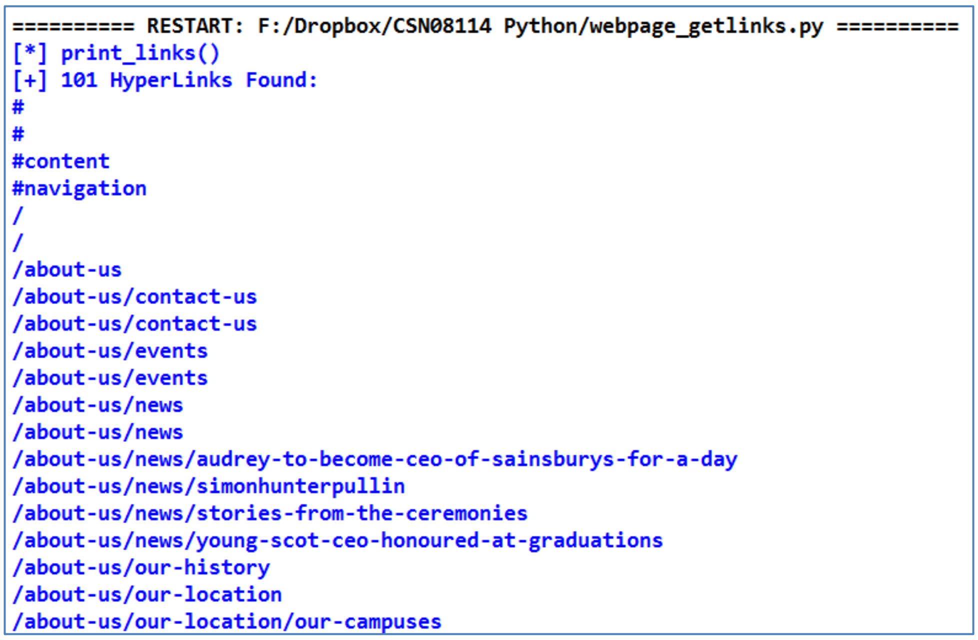
The output should now be similar to the following:



```
========== RESTART: F:/Dropbox/CSN08114 Python/webpage_getlinks.py ==========
[*] print_links()
[+] 101 HyperLinks Found:
<a class="CTABanner" href="/study-with-us/graduation">
<a class="MainLogo" href="/">
<a class="openMenu icon-menu" aria-label="open menu" href="#"></a>
<a href="#" class="advancedToggle">Advanced <span class="icon-triangle-down"><
</a>
<a href="#content" class="screenreader">Skip to main content</a>
<a href="#navigation" class="screenreader">Skip to navigation</a>
<a href="/">
<a href="/about-us/events">
<a href="/about-us/news">
<a href="/about-us/news/audrey-to-become-ceo-of-sainsburys-for-a-day" id="ctl1
ediamondtopcontent_0_ctaLink">
<a href="/about-us/news/simonhunterpullin" id="homepagestandardcontent_4_linkF
ticle">Nursing student turns jail time into an award-winning experience </a></
<a href="/about-us/news/stories-from-the-ceremonies" id="homepagestandardconte
inkSecondArticle">Stories from the autumn ceremonies</a></h1>
```

So far so good. But clearly we are getting not just the link itself, but the whole of the <a> tag that each link is embedded in.

Improve the regular expression pattern further, until you have the same number of links printed out without any other text. The output should be similar to the following:



```
========== RESTART: F:/Dropbox/CSN08114 Python/webpage_getlinks.py ==========
[*] print_links()
[+] 101 HyperLinks Found:
#
#
#content
#navigation
/
/
/about-us
/about-us/contact-us
/about-us/contact-us
/about-us/events
/about-us/events
/about-us/news
/about-us/news
/about-us/news/audrey-to-become-ceo-of-sainsburys-for-a-day
/about-us/news/simonhunterpullin
/about-us/news/stories-from-the-ceremonies
/about-us/news/young-scot-ceo-honoured-at-graduations
/about-us/our-history
/about-us/our-location
/about-us/our-location/our-campuses
```

Tip: Build the pattern by adding characters which are in the valid hyperlinks and don't include characters which are outwith the links. The pattern may need groups, with the group you are interested in being all characters in a valid hyperlink. You may also need non-greedy matching.

Refer back to Lecture 6 and the resources given there for help with regular expressions.

Q: What was the working pattern?

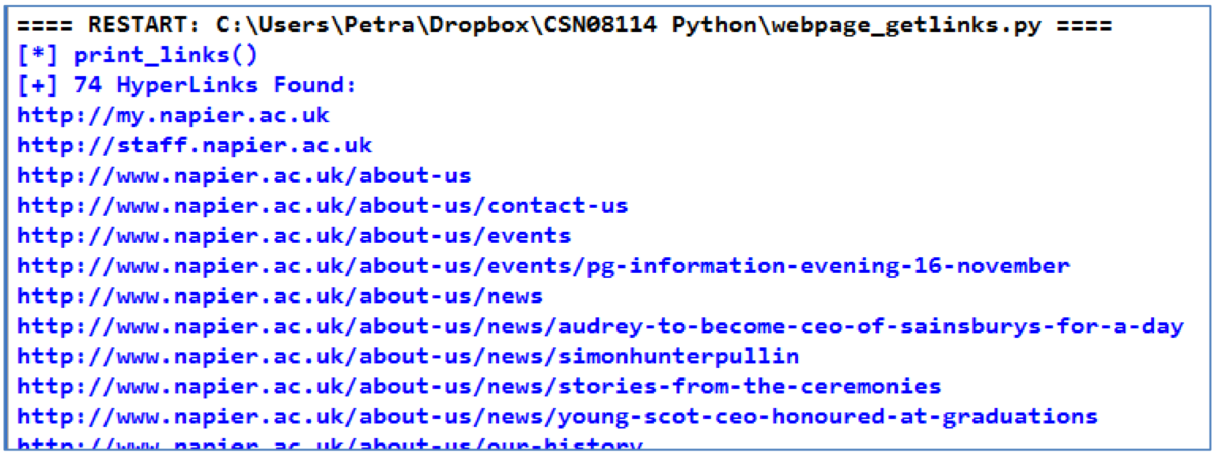
2. Challenge exercise: Print full path for all unique hyperlinks

If we are really picky about our previous solution, we now have some more annoying issues. Firstly, we are getting some hyperlinks repeatedly. Secondly, we are finding relative hyperlinks, but if we wanted to visit them, we would need the full path. Thirdly, there is really no point printing # and / as hyperlinks in their own right. / just refers to the page itself, which we are already reading anyway. Any local hyperlinks that start with # are bookmarks on the same page, so again, we are already reading these.

Your challenge is to fix these issues. For the second problem, you may want to wait until you have worked through the rest of the lab, specifically exercise 8.

Finally, your output should now look similar to the screenshot below.

Remember that the number of links found should also be lower now!

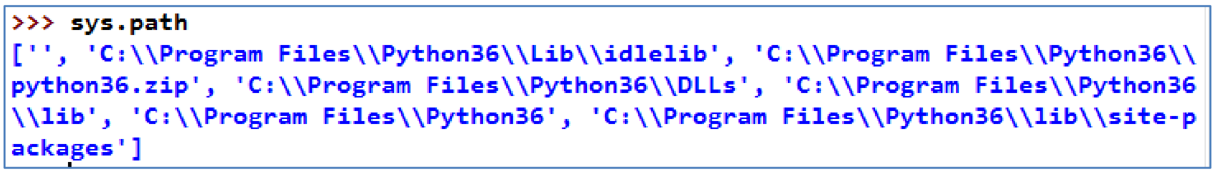


```
==== RESTART: C:\Users\Petra\Dropbox\CSN08114 Python\webpage_getlinks.py ====
[*] print_links()
[+] 74 HyperLinks Found:
http://my.napier.ac.uk
http://staff.napier.ac.uk
http://www.napier.ac.uk/about-us
http://www.napier.ac.uk/about-us/contact-us
http://www.napier.ac.uk/about-us/events
http://www.napier.ac.uk/about-us/events/pg-information-evening-16-november
http://www.napier.ac.uk/about-us/news
http://www.napier.ac.uk/about-us/news/audrey-to-become-ceo-of-sainsburys-for-a-day
http://www.napier.ac.uk/about-us/news/simonhunterpullin
http://www.napier.ac.uk/about-us/news/stories-from-the-ceremonies
http://www.napier.ac.uk/about-us/news/young-scot-ceo-honoured-at-graduations
http://www.napier.ac.uk/about-us/our-history
```

3. Python System Scripting: the sys module

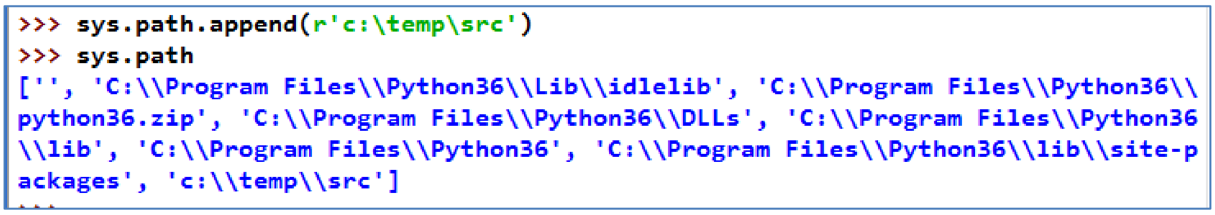
The `sys` standard module provides functions and global variables for interaction with the host operating system. Let's use the help and dir BIFs to check what the sys module can provide for us.

We can list the IDLE/Python search path using **sys.path:**



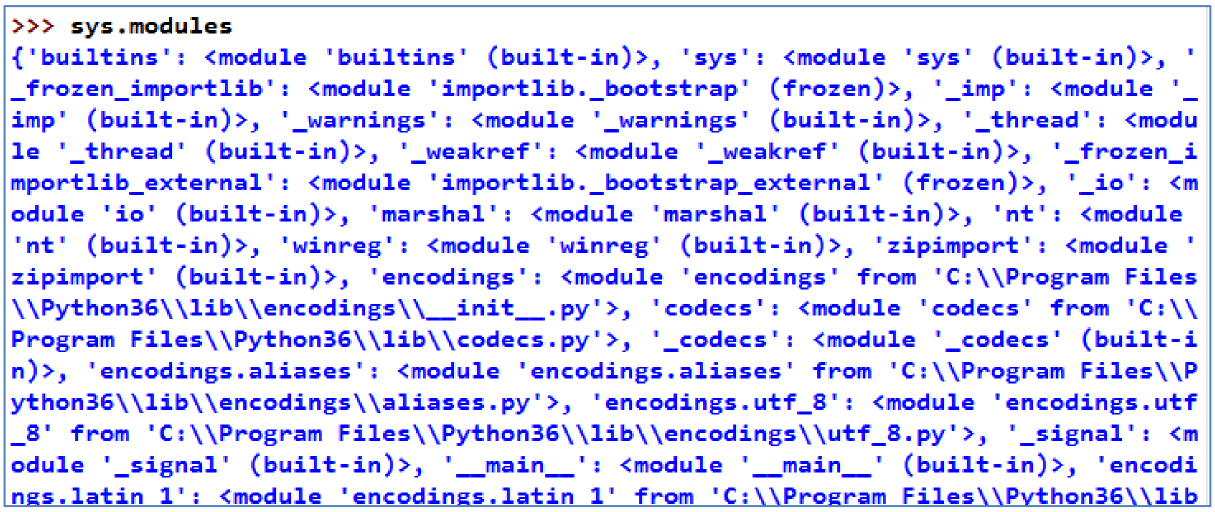
```
>>> sys.path
['', 'C:\\Program Files\\Python36\\Lib\\idlelib', 'C:\\Program Files\\Python36\\
python36.zip', 'C:\\Program Files\\Python36\\DLLs', 'C:\\Program Files\\Python36
\\lib', 'C:\\Program Files\\Python36', 'C:\\Program Files\\Python36\\lib\\site-p
ackages']
```

We can add to the module search path using **sys.path.append()**. Try:



```
>>> sys.path.append(r'c:\temp\src')
>>> sys.path
['', 'C:\\Program Files\\Python36\\Lib\\idlelib', 'C:\\Program Files\\Python36\\
python36.zip', 'C:\\Program Files\\Python36\\DLLs', 'C:\\Program Files\\Python36
\\lib', 'C:\\Program Files\\Python36', 'C:\\Program Files\\Python36\\lib\\site-p
ackages', 'c:\\temp\\src']
```

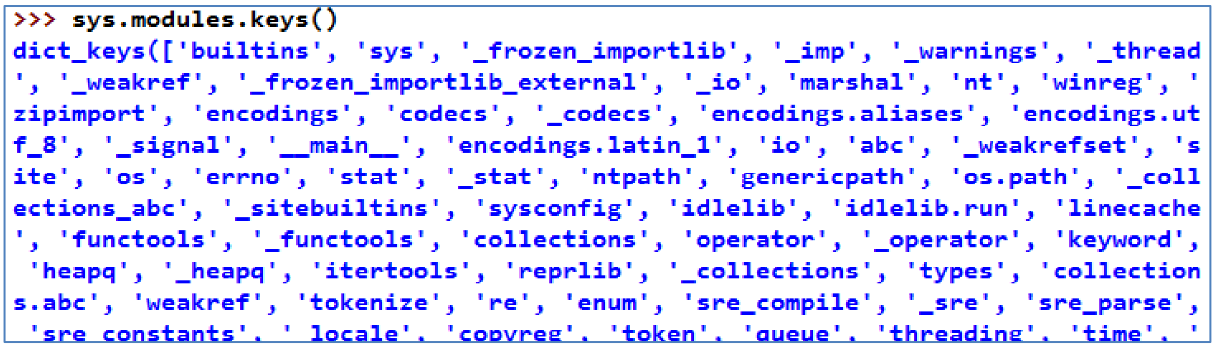
With sys.modules, we can check which modules are currently loaded in Python:



```
>>> sys.modules
{'builtins': <module 'builtins' (built-in)>, 'sys': <module 'sys' (built-in)>, '
_frozen_importlib': <module 'importlib._bootstrap' (frozen)>, '_imp': <module '_
imp' (built-in)>, '_warnings': <module '_warnings' (built-in)>, '_thread': <modu
le '_thread' (built-in)>, '_weakref': <module '_weakref' (built-in)>, '_frozen_i
mportlib_external': <module 'importlib._bootstrap_external' (frozen)>, '_io': <m
odule 'io' (built-in)>, 'marshal': <module 'marshal' (built-in)>, 'nt': <module
'nt' (built-in)>, 'winreg': <module 'winreg' (built-in)>, 'zipimport': <module '
zipimport' (built-in)>, 'encodings': <module 'encodings' from 'C:\\Program Files
\\Python36\\lib\\encodings\\__init__.py'>, 'codecs': <module 'codecs' from 'C:\\
Program Files\\Python36\\lib\\codecs.py'>, '_codecs': <module '_codecs' (built-i
n)>, 'encodings.aliases': <module 'encodings.aliases' from 'C:\\Program Files\\P
ython36\\lib\\encodings\\aliases.py'>, 'encodings.utf_8': <module 'encodings.utf
_8' from 'C:\\Program Files\\Python36\\lib\\encodings\\utf_8.py'>, '_signal': <m
odule '_signal' (built-in)>, '__main__': <module '__main__' (built-in)>, 'encodi
ngs.latin_1': <module 'encodings.latin_1' from 'C:\\Program Files\\Python36\\lib
```

Q: What type of object is sys.modules?

Because sys.modules is a dictionary, we can use sys.modules.keys() to list just the modules loaded without all the additional information

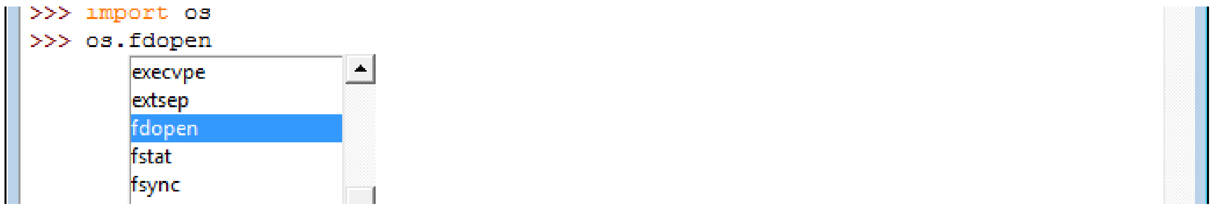


```
>>> sys.modules.keys()
dict_keys(['builtins', 'sys', '_frozen_importlib', '_imp', '_warnings', '_thread
', '_weakref', '_frozen_importlib_external', '_io', 'marshal', 'nt', 'winreg', '
zipimport', 'encodings', 'codecs', '_codecs', 'encodings.aliases', 'encodings.ut
f_8', '_signal', '__main__', 'encodings.latin_1', 'io', 'abc', '_weakrefset', 's
ite', 'os', 'errno', 'stat', '_stat', 'ntpath', 'genericpath', 'os.path', '_coll
ections_abc', '_sitebuiltins', 'sysconfig', 'idlelib', 'idlelib.run', 'linecache
', 'functools', '_functools', 'collections', 'operator', '_operator', 'keyword',
 'heapq', '_heapq', 'itertools', 'reprlib', '_collections', 'types', 'collection
s.abc', 'weakref', 'tokenize', 're', 'enum', 'sre_compile', '_sre', 'sre_parse',
 'sre_constants', '_locale', 'copyreg', 'token', 'queue', 'threading', 'time', '
```

4. Operating System Interaction– The **os** Module

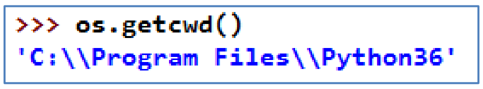
Previously we looked briefly at the Python **os** module for accessing os and directory information. It also contains os functions to deal with shell variables and processes, among other things. Check the help and methods available with help BIF.

Import the os module, and use <CTRL-SPACE>, and scroll through the available functions.



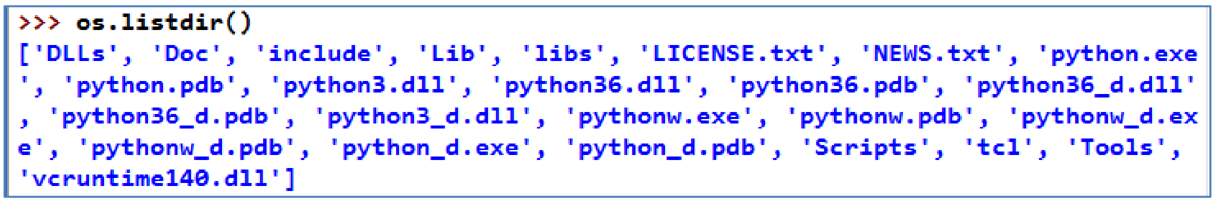
```
>>> import os
>>> os.fdopen
```

Use the **os.getcwd()** function to check the current working directory:



```
>>> os.getcwd()
'C:\\Program Files\\Python36'
```

List the files in the directory with `os.listdir()`:



```
>>> os.listdir()
['DLLs', 'Doc', 'include', 'Lib', 'libs', 'LICENSE.txt', 'NEWS.txt', 'python.exe
', 'python.pdb', 'python3.dll', 'python36.dll', 'python36.pdb', 'python36_d.dll'
, 'python36_d.pdb', 'python3_d.dll', 'pythonw.exe', 'pythonw.pdb', 'pythonw_d.ex
e', 'pythonw_d.pdb', 'python_d.exe', 'python_d.pdb', 'Scripts', 'tcl', 'Tools',
'vcruntime140.dll']
```

Q: What type of object has been returned?

5. Platform Independent Constants

The os module has some platform independent constants which are really useful for creating portable code that works on different operating systems. Try:

```
>>> os.curdir, os.sep, os.linesep, os.pathsep
```

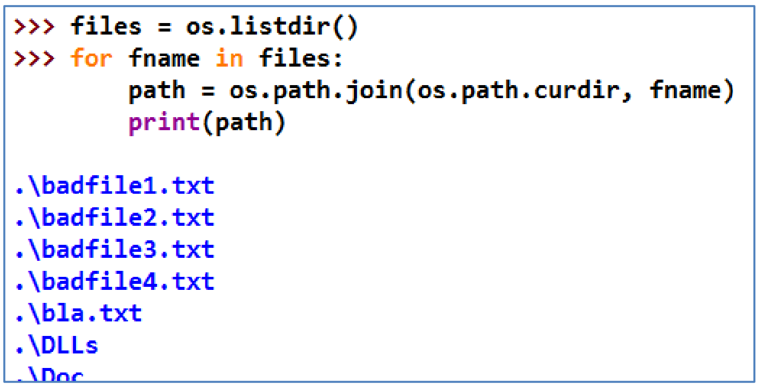
Q: What is(are) the Windows line separating (termination) character(s)?

6. File System Interaction– The `os.path` Module

The `os.path` module is a sub-module within os. You can import it on its own if you don't need the rest of the os module. If you import os, os.path is automatically included. os.path can be used to manipulate directory and file paths of the underlying file system. As with os, scripts written with os.path will be portable across different platforms.

Let's use the file list returned from os.listdir(), and iterate over the files to do something with each file using the **os.path** module functions.

In the interpreter shell, try:



```
>>> files = os.listdir()
>>> for fname in files:
        path = os.path.join(os.path.curdir, fname)
        print(path)

.\badfile1.txt
.\badfile2.txt
.\badfile3.txt
.\badfile4.txt
.\bla.txt
.\DLLs
.\Doc
```

The path we have printed here is the **relative path,** relative to the current directory. This is because os.path.curdir = '.', i.e. it gives the relative path to the current directory.

Often we need the absolute path (or full path) rather than the relative path. In Windows, this starts with the drive letter, e.g. C:\Program Files\Python37\include. The absolute path can be generated from the relative path using the **abspath()** method.

Replace the line `print(path)` with:

```
>>> print (os.path.abspath(path))
```

Q: Does the output now show absolute paths? Y/N

Q: Why might the absolute path be needed?

We can check if a file or directory exists using **os.path.exists()**.

Try:

```
>>> os.path.exists(r'c:\nonexistent')
```

Q: Does the dir exist?

Q: Why does the lower case c: not throw an error?

Try:

```
>>> os.path.exists(os.curdir)
```

Q: Does the dir exist?

We can use the functions **os.path.isdir(),** and **os.path.isfile()** to check whether we are dealing with a file or a directory:

```
>>> os.path.isfile(os.curdir)
>>> os.path.isdir(os.curdir)
```

Q: Is os.curdir a file or directory?

In the IDLE Shell, using **File>New Window**, create a script **ls.py**, with the code below:

```
# Script:  ls.py
# Desc:    List files and subdirectories of current directory.
# Modified: Oct 2017, Nov18 (PEP8)
#
import os
```

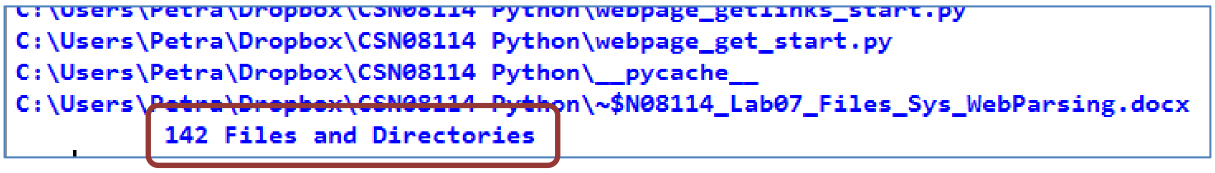
```
curdir = os.curdir  # dir to list files for

# print details for all objects in dir
files = os.listdir(curdir)
for filename in files:
    path = os.path.join(curdir, filename)
    print(os.path.abspath(path))
```

This small script lists all objects (files and subdirectories) in the current directory with absolute paths.

Test from the IDLE shell, by running the module using **Run>Run Module.**

***Optional** enhancements*

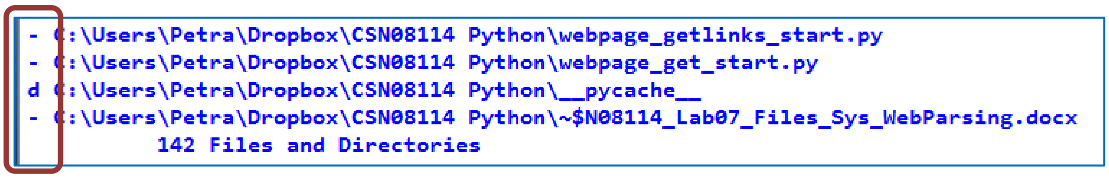
Add a line to your script to print a summary at the bottom of the output, like this:



```
C:\Users\Petra\Dropbox\CSN08114 Python\webpage_getlinks_start.py
C:\Users\Petra\Dropbox\CSN08114 Python\webpage_get_start.py
C:\Users\Petra\Dropbox\CSN08114 Python\__pycache__
C:\Users\Petra\Dropbox\CSN08114 Python\~$N08114_Lab07_Files_Sys_WebParsing.docx
         142 Files and Directories
```

Q: How many files in your directory?

Improve the ls.py script further to show which items are files and which are directories. Let's do this linux-style by creating a prefix character, using – for a file, or d for a directory. The output should now be similar to the following:



```
- C:\Users\Petra\Dropbox\CSN08114 Python\webpage_getlinks_start.py
- C:\Users\Petra\Dropbox\CSN08114 Python\webpage_get_start.py
d C:\Users\Petra\Dropbox\CSN08114 Python\__pycache__
- C:\Users\Petra\Dropbox\CSN08114 Python\~$N08114_Lab07_Files_Sys_WebParsing.docx
          142 Files and Directories
```

7. **Optional** File Sizes and MAC times

The .getsize() method returns the size of a file/directory. (But note that this value is meaningless for directories). From the interpreter, try the following:

```
>>> print(os.path.getsize(r'c:\Program Files\Python37\LICENSE.txt'))
>>> print(os.path.getsize(r'c:\Program Files\Python37\python.exe'))
```

Q: What sizes are the two files?

Q: What unit does .getsize use? (Hint: look up the Python docs for os.stat - google is your friend)

os.path also has methods to get the MAC (modified, accessed, created) timestamps for directories and files. For example, the **os.path.getctime()** function gets the created timestamp.

Try:

```
>>> path = r'c:\Program Files\Python37\python.exe'
>>> print (os.path.getctime(path))
```

(if this file doesn't exist, just replace the path with the full path to any file that you know exists)

Q: What is the creation timestamp for the file?

Q: In what units is the timestamp returned? (Hint: look up the Python docs for os.stat - google is your friend)

Two similar functions, .getmtime() and .getatime() can be used to show modified and accessed time stamps respectively. As different file systems store MAC (modified, accessed, created) times differently, the results you get and the format will depend on the underlying file system. If your course includes a digital forensics module, you will learn more about this in that module.

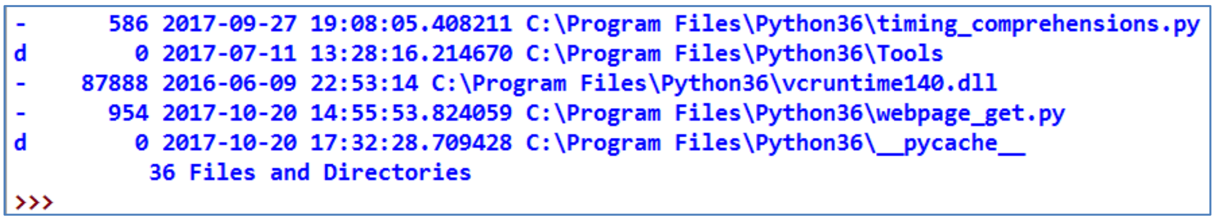
The timestamp can be changed into a readable date and time using the datetime module.

```
>>> timestamp = os.path.getctime(path)
>>> import datetime
>>> print(datetime.datetime.fromtimestamp(timestamp))
```

Q: What is the created date and time for the file in human readable form?

Add code to your script so that it also displays the size and last modified date and time for each item in the listing. Format the size to 8d (we covered this in an earlier lab) so that it's aligned more neatly.

Your output should now look similar to the following:

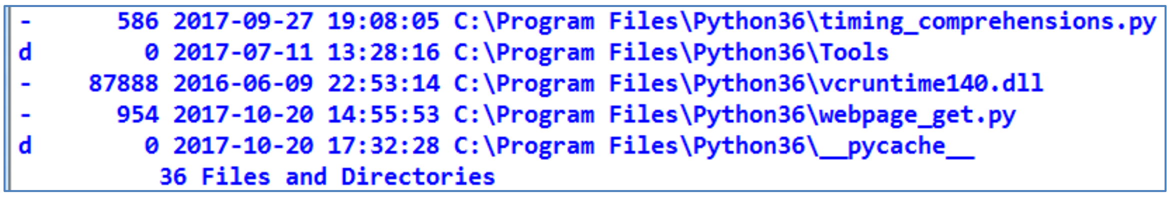


```
-      586 2017-09-27 19:08:05.408211 C:\Program Files\Python36\timing_comprehensions.py
d        0 2017-07-11 13:28:16.214670 C:\Program Files\Python36\Tools
-    87888 2016-06-09 22:53:14 C:\Program Files\Python36\vcruntime140.dll
-      954 2017-10-20 14:55:53.824059 C:\Program Files\Python36\webpage_get.py
d        0 2017-10-20 17:32:28.709428 C:\Program Files\Python36\__pycache__
          36 Files and Directories
>>>
```

Annoyingly, some files and directories have milliseconds as part of their timestamp (after the dot). You can get rid of this by converting the timestamp to a string (using the string function), then splitting on the '.' and choosing only index[0] of the resulting list:

```
>>> timestamp = str(timestamp).split('.')[0]
```

Finally, this should give you neat output like this:



```
-      586 2017-09-27 19:08:05 C:\Program Files\Python36\timing_comprehensions.py
d        0 2017-07-11 13:28:16 C:\Program Files\Python36\Tools
-    87888 2016-06-09 22:53:14 C:\Program Files\Python36\vcruntime140.dll
-      954 2017-10-20 14:55:53 C:\Program Files\Python36\webpage_get.py
d        0 2017-10-20 17:32:28 C:\Program Files\Python36\__pycache__
          36 Files and Directories
```

8. Joining and Splitting Paths

In os.path, there are also functions for splitting directory and filename. Try the following:

```
>>> path1 = r'C:\Program Files\Python37\python.exe'
>>> os.path.dirname(path1)
>>> os.path.basename(path1)
```

Q: What does the **dirname()** method return?

Q: What does the **basename()** method return?

Also the following can be useful if file types are needed:

```
>>> os.path.splitext(path1)
```

Q: What does the **splitext()** method return?

Finally, the platform independent constant os.sep can be used to split the path also. Try:

```
>>> path1.split(os.sep)
```

Q: What does path1.split(os.sep) return?

9. External Data – Writing Data to Files

A we have seen before, the `open()` BIF opens and returns a file object. The built in `file` object contains the basic file handling methods to read data from files, such as read(). It also has functions to write data to files.

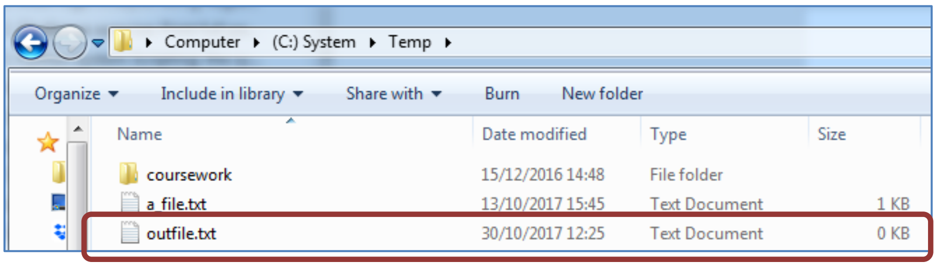
From the Python Interpreter shell, use the BIF **open()** to open a file in write mode using the mode='w' argument. This returns a file object **outfile**.

Check that C:\temp exists and create it if necessary. Then try:

```
>>> outfile = open(r'c:\temp\outfile.txt', 'w')
```

The file outfile.txt should be automatically created with Size=0 by the open() BIF.

Q: Has the file been created in c:\temp?

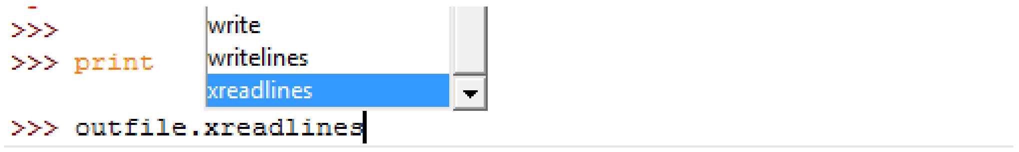


Computer ▸ (C:) System ▸ Temp ▸
Organize ▾ Include in library ▾ Share with ▾ Burn New folder

Name	Date modified	Type	Size
coursework	15/12/2016 14:48	File folder	
a file.txt	13/10/2017 15:45	Text Document	1 KB
outfile.txt	30/10/2017 12:25	Text Document	0 KB

The easiest way to write to the file is to use the standard library **print** BIF. It writes to stdout by default, but it has a **file** argument which we can use to write to a file.

Check the methods available on the file object for writing using **.CTRL+SPACE**:



```
>>>
>>> print
>>> outfile.xreadlines
```

Q: Which methods might be useful to write data to the file?

We can use the **write()** method to write lines to the file.

```
>>> outfile.write('test line\n')
>>> outfile.write('test line2\n')
>>>
```

Q: Check the contents of the file. Have the lines been written?

It looks like the lines have not been written yet. This is because the data is held in a buffer until the buffer is flushed or the file closed. Try:

```
>>> outfile.close()
```

Q: Check the contents of the file. Have the lines been written now?

Open the file again in write mode, and write the following lines:

```
>>> outfile.write('test line3\n')
>>> outfile.write('test line4\n')
```

Close the file and check the contents.

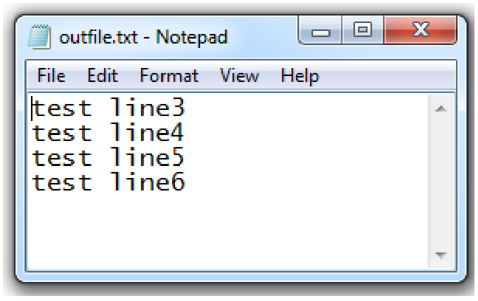
Q: Check the contents of the file. How many lines does it contain now?

The file 'w' write mode clears the file of its old contents if the file exists already. This is known as **clobbering** a file. To **append** to a file, open it using append mode; mode='a'.

Try:

```
>>> outfile = open(r'c:\temp\outfile.txt', 'a')
>>> outfile.write('test line5\n')
>>> outfile.write('test line6\n')
>>> outfile.close()
```

Q: Check the contents of the file. How many lines does it contain now?



```
test line3
test line4
test line5
test line6
```

10. **Optional** Writing directory listing to a file

Let's combine some of the exercises above.

Copy the ls.py script. Then change the copy so it writes a log file instead of printing the info to the screen.

The main change you need to make to your existing code is to replace print statements with logfile.write statements. The main difference between these two is that print automatically adds a new line, while write() does not.

So, the line

```
print(f'{size:8d} {time} {os.path.abspath(path)}')
```

should be changed to:

```
logfile.write(f'{size:8d} {time} {os.path.abspath(path)}\n')
```

As we are opening and writing files, we need to ensure that the file is always closed, even if an error occurs. Therefore you should wrap your code in a try: block.

Your new code should look like this:

```
try:
    logfile = open(r'c:\temp\lslog.txt', 'w')

    # your existing code here, but change print statements to
    # logfile.write() statements

except IOError as err:
    print(f'File Error: {IOError}')
finally:
    if 'logfile' in locals():
        logfile.close()
```

Q: Does your ls.py script now write the file details listing to the log file in c:\temp?